BEE 4750 Lab 2: Uncertainty and Monte Carlo

Name: Jonathan Marcuse

ID: jrm564

> **Due Date**
>
> Friday, 9/22/23, 9:00pm

Setup

The following code should go at the top of most Julia scripts; it will load the local package environment and install any needed packages. You will see this often and shouldn't need to touch it.

In [ ]:
```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
  Activating project at `~/Desktop/Cornell/Fall2023/BEE4750/lab-02-jrmarcuse
`
```

In [ ]:
```
using Random # random number generation
using Distributions # probability distributions and interface
using Statistics # basic statistical functions, including mean
using Plots # plotting
```

Introduction

In this lab, you will use Monte Carlo analysis to estimate the expected winnings for a couple of different games of chance.

Monte Carlo methods involve the simulation of random numbers from probability distributions. In an environmental context, we often propagate these random numbers through some more complicated model and then compute a resulting statistic which is relevant for assessing performance or risk, such as an average outcome or a particular quantile.

Julia provides a common interface for probability distributions with the `Distributions.jl` package. The basic workflow for sampling from a distribution is:

1. Set up the distribution. The specific syntax depends on the distribution and what parameters are required, but the general call is the similar. For a normal distribution or a uniform distribution, the syntax is

   ```
   # you don't have to name this "normal_distribution"
   # μ is the mean and σ is the standard deviation
   normal_distribution = Normal(μ, σ)
   # a is the upper bound and b is the lower bound; these can be set to +Inf or -Inf for an unbounded distribution in one or both directions.
   uniform_distribution = Uniform(a, b)
   ```

   There are lots of both univariate and multivariate distributions, as well as the ability to create your own, but we won't do anything too exotic here.

2. Draw samples. This uses the `rand()` command (which, when used without a distribution, just samples uniformly from the interval $[0, 1]$.) For example, to sample from our normal distribution above:

   ```
   # draw n samples
   rand(normal_distribution, n)
   ```

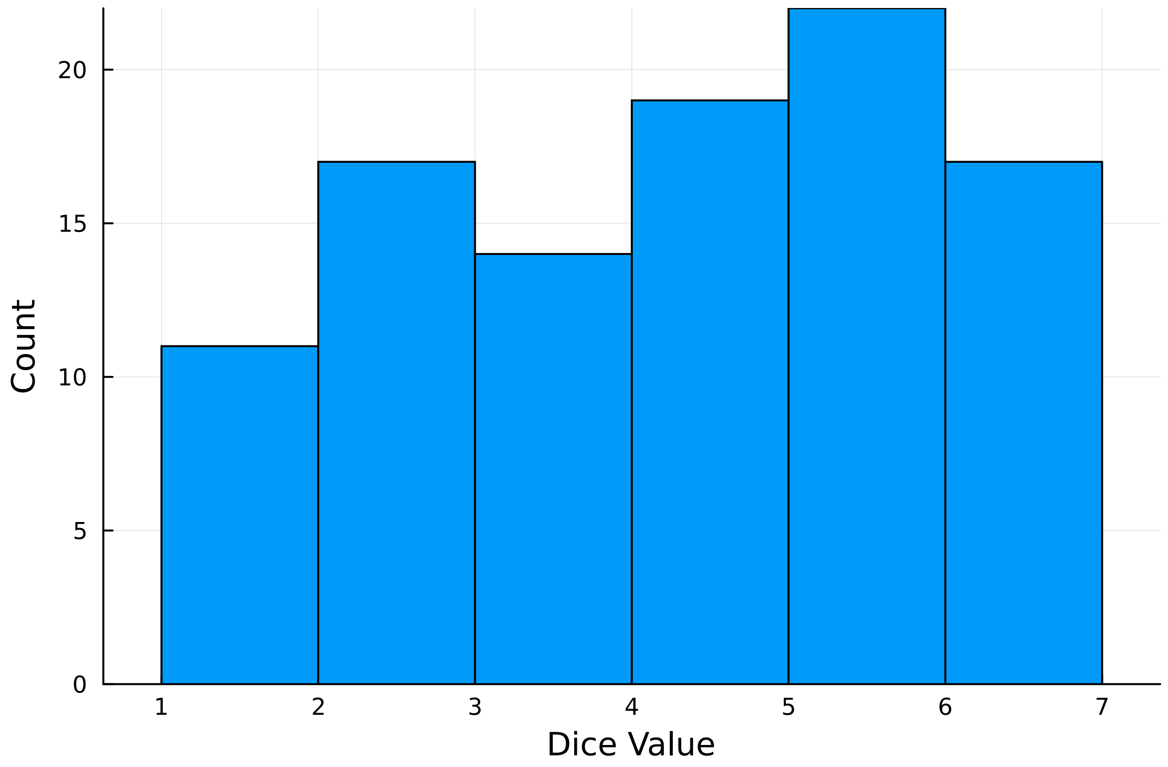
Putting this together, let's say that we wanted to simulate 100 six-sided dice rolls. We could use a Discrete Uniform distribution.

```
In [ ]: # can generate any integer between 1 and 6
        dice_dist = DiscreteUniform(1, 6)
        dice_rolls = rand(dice_dist, 100) # simulate rolls
```

```
100-element Vector{Int64}:
 6
 5
 4
 3
 6
 4
 3
 4
 5
 1
 ⋮
 2
 5
 6
 4
 1
 6
 4
 5
 4
```

And then we can plot a histogram of these rolls:

In [ ]:
```
histogram(dice_rolls, legend=:false, bins=6)
ylabel!("Count")
xlabel!("Dice Value")
```



Remember to:

- Evaluate all of your code cells, in order (using a `Run All` command). This will make sure all output is visible and that the code cells were evaluated in the correct order.
- Tag each of the problems when you submit to Gradescope; a 10% penalty will be deducted if this is not done.

Exercises (10 points)

In Problem 1, you will compute the probability of getting a specific combination of multiple dice rolls. The focus will be on understanding how the Monte Carlo estimate changes based on the number of simulations.

In Problem 2, we will implement the culmination of every episode of the long-running game show The Price Is Right: the Showcase. You will be asked to make a plot of expected winnings by bid for a particular distribution of prize values.

You should always start any computing with random numbers by setting a “seed,” which controls the sequence of numbers which are generated (since these are not *really* random, just “pseudorandom”). In Julia, we do this with the `Random.seed!()` function.

```
Random.seed!(1)
```

```
    TaskLocalRNG()
```

It doesn’t matter what seed you set, though different seeds might result in slightly different values. But setting a seed means every time your notebook is run, the answer will be the same.

> **Seeds and Reproducing Solutions**
>
> If you don’t re-run your code in the same order or if you re-run the same cell repeatedly, you will not get the same solution. If you’re working on a specific problem, you might want to re-use `Random.seed()` near any block of code you want to re-evaluate repeatedly.

Problem 1 (5 points)

We want to know the probability of getting at least an 11 from rolling three fair, six-sided dice (this is actually an old Italian game called *passadieci*, which was analyzed by Galileo as one of the first examples of a rigorous study of probability).

Problem 1.1 (1 point)

Write a function called `passadieci()` to simulate this game, which will take as an input the number of realizations and output a vector of the sum of the three dice rolls for each realization.

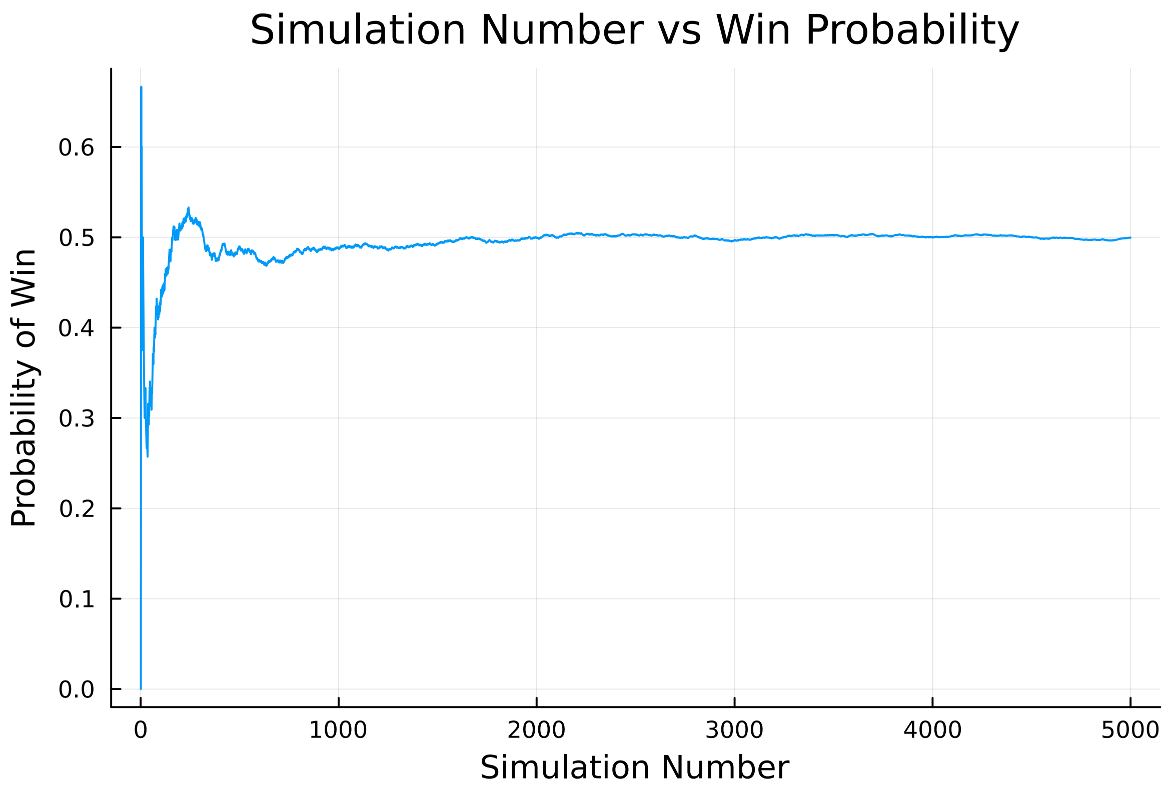
```
In [ ]: #This function simulates a game of passadieci,
            #and the input is the number of times it is played

        function passadieci(realizations)
            output_vec=[]; #initialize the output vector
            for i=1:realizations
                dice_dist = DiscreteUniform(1, 6) # can generate any integer
                                                  #between 1 and 6
                dice_rolls = rand(dice_dist, 3) # simulate rolls
                dice_3_sum = sum(dice_rolls[:]);
                    # sum up the 3 rolls in this iteration
                push!(output_vec,dice_3_sum)
                #append the sum of the 3 dice rolls to the passadieci vector
            end
            return output_vec
        end
```

```
passadieci (generic function with 1 method)
```

Problem 1.2 (2 points)

Generate 5,000 simulations of the game using your `passadieci()` function. Plot how the computed probability of winning the game changes as the number of simulations increases (you can do this by computing the frequency of wins for each additional simulation).

In [ ]:
```
#This code plots the probability changes of winning the passadieci
    #as a function of n, the number of times the game is played

#make pvec a vector of 5000 outputs from the passadieci function
pvec = passadieci(5000);
#initialize variables
wins=0;
prob=[];
#for loop to assess if each vector unit is a win or a loss,
    #and then create a new vector of the cumulative probability
for i=1:5000
    if pvec[i]>=11
        wins=wins+1;
        push!(prob, wins/i);
    else
        push!(prob, wins/i);
    end
end
#keep data consistent
Random.seed!(1)

#plot the results to reveale convergence
plot(prob, legend=false, xlabel="Simulation Number",
ylabel="Probability of Win",title="Simulation Number vs Win Probability")
```



Problem 1.3 (2 point)

Based on your plot from Problem 1.2, how many simulations were needed for the win probability estimate to converge? What did you notice from your plot about the estimates prior to convergence?

About 1000 simulations, and beforehand the probability was much more volatile and changed very quickly. Later it is more steady and almost a straight line close to 0.5

Problem 2 (5 points)

The Showcase is the final round of every episode of The Price is Right, matching the two big winners from the episode. Each contestant is shown a "showcase" of prizes, which are usually some combination of a trip, a motor vehicle, some furniture, and maybe some other stuff. They then each have to make a bid on the retail price of the showcase. The rules are:

- an overbid is an automatic loss;
- the contest who gets closest to the retail price wins their showcase;
- if a contestant gets within $250 of the retail price and is closer than their opponent, they win both showcases.

Your goal is to find a wager which maximizes your expected winnings, which we may as well call utility, based on your assessment of the probability of your showcase retail price. We'll assume that the distribution of all showcases offered by the show is given as truncated normal distribution, which means a normal distribution which has an upper and/or lower bound. `Distributions.jl` makes it easy to specify truncations on any distribution, not just normal distributions. For example, we'll use this distribution for the showcase values:

```
showcase_dist = truncated(Normal(31000, 4500), lower=5000,
upper=42000)
```

```
Truncated(Normal{Float64}(μ=31000.0, σ=4500.0); lower=5000.0,
upper=42000.0)
```

Problem 2.1 (3 points)

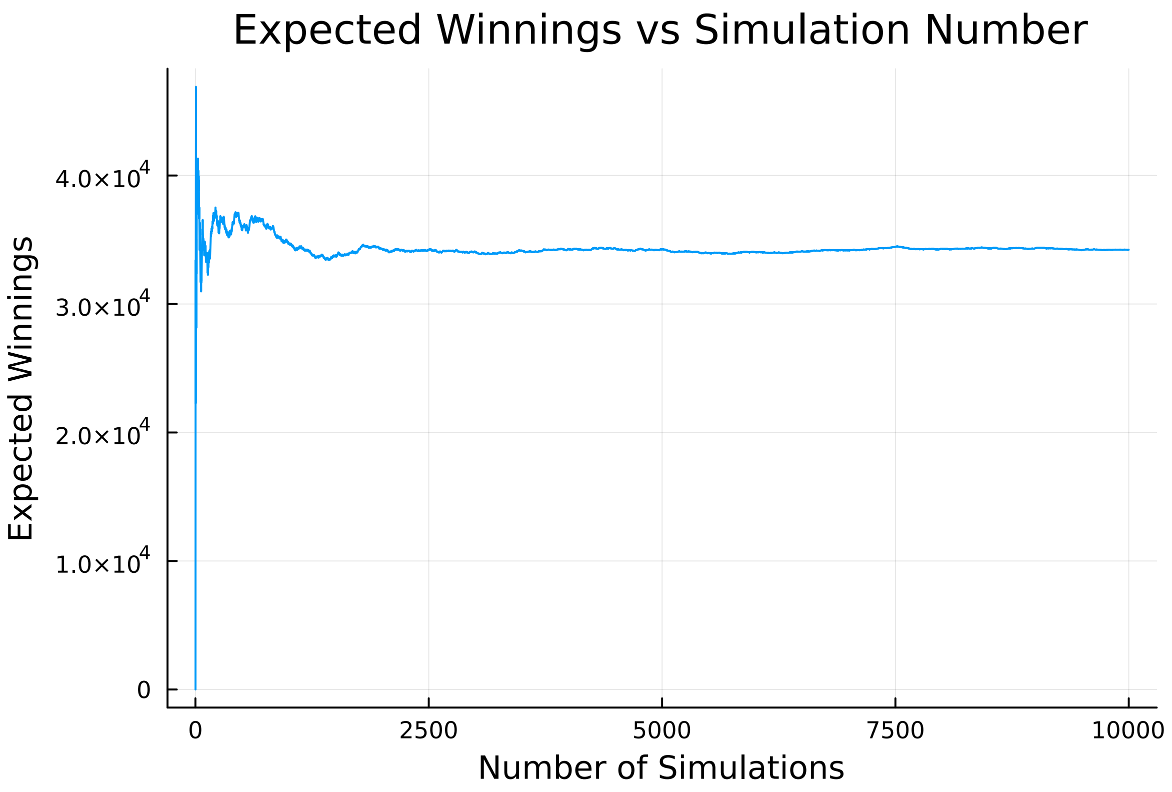
Write a function `showcase()` which takes in a bid value and uses Monte Carlo simulation to compute the expected value of the winnings. Make the following assumptions about your expected winnings if you don't overbid:

- If you win both showcases, the value is the double of the single showcase value.
- If you did not win both showcases but bid under the showcase value, the probability of being outbid increases linearly as the distance between your bid and the value increases (in other words, if you bid the exact value, you win with probability 1, and if you bid $0, you win with probability 0).

How did you decide how many samples to use within the function?

In [ ]:

```
showcase_dist = truncated(Normal(31000,4500), lower=5000, upper=42000)
#initialize vector of winnings
won=[];
bid_value=31000;

for i=1:10000
    #Initialize winnings variable at 0
    winnings=0;
    final_showcase_value = rand(showcase_dist);

    #Gets the probability of winning to be scaled to win
    probability_of_win = 1-
    ((final_showcase_value-bid_value)/final_showcase_value);

    #This set of if statements determines the winnings based on your guess
    if bid_value<=final_showcase_value
        if final_showcase_value-bid_value-bid_value<=250
            winnings=2*final_showcase_value;
        elseif rand()<probability_of_win
            #rand has an even prob of choosing anythng between 0 and 1
            #so if it picks something less than the probability we
            #can call it a win
            winnings=final_showcase_value;
        end
    end
    #append the winnings of this run to the won vector
    push!(won,winnings)
end

#Calculate running means of won vector
running_means=[]
for i=1:length(won)
    push!(running_means,mean(won[1:i]))
end

plot(running_means, legend=false, xlabel="Number of Simulations",
ylabel="Expected Winnings",title="Expected Winnings vs Simulation Number")
```



I am choosing a sample amount of 2500 because the plot above shows the expected value is very variable under 1000 and then slowly increases until it reaches 5000 and then it levels off.

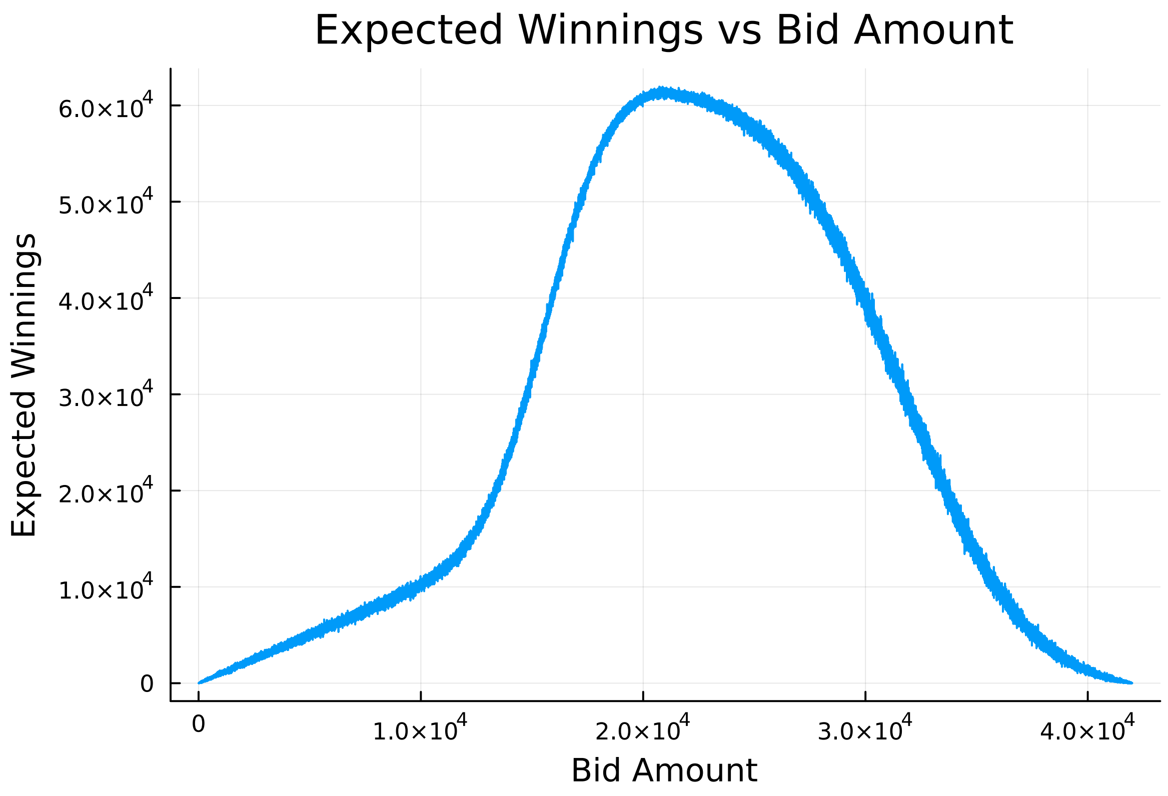
In [ ]:
```
#This function determines the expected value of the winnings based
#on your chosen bid
function showcase(bid_value)
    #Create the distribution
    showcase_dist = truncated(Normal(31000,4500),
    lower=5000, upper=42000)
    #initialize vector of winnings
    won=[];
    for i=1:2500
        #Initialize winnings variable at 0
        winnings=0;
        final_showcase_value = rand(showcase_dist)

        #Gets the probability of winning to be scaled to win
        probability_of_win = 1-
        ((final_showcase_value-bid_value)/final_showcase_value);

        #This set of if statements determines the winnings based
        #on your guess
        if bid_value<=final_showcase_value
            if final_showcase_value-bid_value-bid_value<=250
                winnings=2*final_showcase_value;
            elseif rand()<probability_of_win
                #rand has an even prob #of choosing anythng between
                #0 and 1 so if it picks something less than the
                #probability we can call it a win
                winnings=final_showcase_value;
            end
        end
        #append the winnings of this simulation to the won vector
        push!(won,winnings)
    end
    return mean(won)
end
```

```
showcase (generic function with 1 method)
```

Problem 2.2 (2 points)

Plot the expected winnings for bids ranging from $0 to 42,000. What do you notice?

In [ ]:
```
using Plots
#make a vector of the expected win value based on bid value
expected_win = [showcase(i) for i in 0:42000]

plot(expected_win,xlabel="Bid Amount",ylabel="Expected Winnings",
title="Expected Winnings vs Bid Amount",legend=false)
```



The winnings are fairly steadily increasing for low bid values and then begins to sharply increase around 13,000 dollars as the probability of winning increases while the probability of winning both showcases is also increasing. The expected earnings increases for a maximum around 21,000 dollars and then decreases thereafter as the probability of bidding more than the showcase amount becomes very high.

References

Put any consulted sources here, including classmates you worked with/who helped you.